1

Project LINQ

COPYRIGHTED MATERIAL

I often hear the questions, ''What is LINQ?,'' ''What does it do?,'' and ''Why do we need it?'' The answer to the first question (and subsequently the other two questions) is that the Language Integrated Query (LINQ) is a set of standard query operators that provide the underlying query architecture for the navigation, filtering, and execution operations of nearly every kind of data source, such as XML (using LINQ to XML, previously known as XLINQ), relational data (using LINQ to SQL, previously known as DLINQ), ADO.NET DataSets (using LINQ to DataSet), and in-memory collections.

The best way to begin understanding this wonderful new technology is to take a look at some history and background on how and why LINQ came to be.

Although the public first became aware of LINQ early in the fall of 2005, LINQ had been in development since early 2003. The overall LINQ goal was to make it easier for developers to interact with SQL and XML, primarily because there exists a disconnect between relational data (databases), XML, and the programming languages that communicate with (that is, work with) each of them.

Most developers understand the concept of object-oriented (OO) programming and its related technologies and features, such as classes, methods, and objects. Object-oriented programming has evolved tremendously over the past 10 years or so, but even in its current state, there's still a gap when using and integrating OO technology with information that is not natively defined or inherent to it.

For example, suppose that you want to execute a T-SQL query from within your C# application. It would look something like this:

```
private void Form1_Load(object sender, EventArgs e)
{
    string ConnectionString = @"Data Source=(local);
     Initial Catalog=AdventureWorks;UID=sa;PWD=yourpassword";
    using (SqlConnection conn = new SqlConnection(ConnectionString))
    {
        conn.Open();
        SqlCommand cmd = conn.CreateCommand();
```

```
        cmd.CommandType = CommandType.Text;
        cmd.CommandText = "SELECT LastName, FirstName FROM
 Person.Contact";
        using (SqlDataReader rdr = cmd.ExecuteReader())
        {
            / / do something
        }
    }
}
```

If you wanted to use the same code to execute a stored procedure that takes one or more parameters, it might look like this:

```
private void Form1_Load(object sender, EventArgs e)
{
    string ConnectionString = @"Data Source=(local);
      Initial Catalog=AdventureWorks;UID=sa;PWD=yourpassword";
    using (SqlConnection conn = new SqlConnection(ConnectionString))
    {
        conn.Open();
        SqlCommand cmd = conn.CreateCommand();
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.CommandText = "uspGetBillOfMaterials";
        cmd.Parameters.Add("@StartProductID", SqlDbType.Int).Value =
324;
        cmd.Parameters.Add("@CheckDate", SqlDbType.DateTime).Value =
"07/10/2000";
        using (SqlDataReader rdr = cmd.ExecuteReader())
        {
            // do something
        }
    }
}
```

While you and I have probably coded something like this many, many times, it isn't ''friendly'' on several levels. First, you are combining two languages into one. You have the language you are coding (in this case C#), plus you have the SQL language in quotation marks, which is not understood in the context of .NET. With the .NET language you have IntelliSense, but you don't get IntelliSense in the embedded SQL syntax.

More importantly, however, there is no compile-time type checking, which means you can't tell if something is broken until run time. Every line of code has to be QA'd just to see if it even begins to work.

Microsoft also packed a lot of features into the .NET Framework that enable developers to work with XML. The .NET Framework contains the `System.Xml` namespace and other supporting namespaces, such as `System.Xml.XPath`, `System.Xml.Xsl`, and `System.Xml.Schema`, which provide a plethora of functionality for working with XML. The namespaces contain many classes and methods that make up the XML .NET API architecture. The main classes are the `XmlDocument`, `XmlReader`, and `XmlWriter`.

To add to the complexity of working with different technologies, parsing an XML document isn't the easiest thing to do, either. Your tools of choice to work with XML are the Document Object Model (DOM),

XQuery, or Extensible Stylesheet Language Transformations (XSLT). For example, to read an XML document using existing technology, you would need to do something like the following:

```
XmlTextReader rdr = new XmlTextReader("C:\Employees.Xml");
while (rdr.Read())
{
    XmlNodeType nt = rdr.NodeType;
    Switch (nt)
    {
        case XmlNodeType.Element:
            break;

        case XmlNodeType.Attribute:
            break;
        case XmlNodeType.Comment:
            break;

        case XmlNodeType.Whitespace:
            break;

     }
}
```

That's a lot of code just to read an XML document (and it isn't even complete). Writing XML isn't any less confusing, as illustrated here:

```
XmlTextWriter wrt = new XmlTextWriter("C:\Employees.Xml");
wrt.WriteStartDocument;
wrt.WriteComment("This is an example");
wrt.WriteStartElement("Employees");
wrt.WriteStartElement("Employee");
wrt.WriteStartElement("FirstName");
wrt.WriteString("Scott");
wrt.WriteEndElement();
wrt.WriteEndElement();
wrt.WriteEndElement();
```

Visually, you don't know if this will work until you compile the project. Likewise, it is hard to see what the resulting XML will look like.

XML is great and its use continues to grow; you can expect it to be around for a long time. Yet, truth be told, XML is still hard to work with.

In dealing with these hurdles, Microsoft considered two paths. The first path would have required the company to build specific XML or relational data features into each programming language and run-time. That would be a major undertaking and an even bigger hassle to maintain. The second option was to add more general-purpose query capabilities into the .NET Framework—in other words, a framework of all-purpose querying facilities built into the .NET Framework that both C# and VB.NET could easily take advantage of.

Luckily, Microsoft chose the later option, creating a unified query experience across objects, XML, collections, and data. It accomplished that by taking query set operations, transforms, and constructs and bringing them to the surface, making them high-level concepts within the .NET Framework (for example,

on the same level as objects and classes). So, you can now enjoy the benefits of a single declarative pattern that can be expressed in any .NET-based programming language.

The result of making these set operations, transforms, and constructs first-class operations is a set of methods called the standard query operators. These operators provide query capabilities that include sorting, filtering, aggregation, and projection over a large number of different data sources. The standard query operators are the focus of Chapter 4, ''LINQ Standard Query Operators.''

Think about it for a minute. A single set of query operators that work within any .NET-based programming language, enabling you to write a query against a database, XML, or an in-memory array using the same syntax? How cool is that? And you get the added benefit of IntelliSense and compile-time type checking! Somebody pinch me.

To illustrate this great technology, take a look at an example that queries the directories of your C drive and writes them to a list box:

```
DirectoryInfo di = new DirectoryInfo("C:\\");
var dirQuery =
    from dir in di.GetDirectories()
    orderby di.Name
    select new { dir.Name} ;

foreach {var item in dirQuery)
    listBox1.Items.Add(item.Name);
```

This code uses some of the standard query operators to create a LINQ query. In essence, Microsoft has taken the concept of query set operations and made them first-class operations within the .NET Framework.

Here's another example. This one queries all the system processes on your PC using the `Process` class, but notice that it uses the same query syntax as the previous example:

```
var procQuery =
    from proc in Process.GetProcesses()
    orderby p.WorkingSet64 descending
    select new { p.Id, p.ProcessName, p.WorkingSet64} ;

foreach (var item in procQuery)
    ListBox1.Items.Add(item.Id + "   " +
        item.ProcessName + "   " +
        item.WorkingSet64);
```

When you run this code, all the processes on your system will be listed in descending order by memory usage.

Simply put, LINQ enables you to query anything that implements the `IEnumerable<T>` interface. If you can loop through the contents using the `foreach` statement, then you can query it using LINQ.

The following example illustrates how LINQ works querying relational data, using a database as the source of data.

```
var conQuery =
    from c in contact
    where c.FirstName.StartsWith("S")
    orderby c.LastName
    select new { c.FirstName, c.LastName, c.EmailAddress} ;

foreach (var item in conQuery)
    ListBox1.Items.Add(item.FirstName + "   " +
        item.LastName + "   " +
        item.EmailAddress);
```

This previous example queries the `Person.Contact` table in the AdventureWorks database for all contacts whose first name starts with the letter ''S''.

The purpose of LINQ is to provide developers with the following benefits:

- A simplified way to write queries.
- Faster development time by removing run-time errors and catching errors at compile time.
- IntelliSense and debugging support for LINQ directly in the development language.
- Closing the gap between relational data and object-oriented development.
- A unified query syntax to use regardless of the source of data.

What is important to notice is the same syntax that you used to query the system processes was used query a SQL data source. Both of these topics will be discussed in much more detail, including how to easily connect and map to the source database.

So, with that primer, this chapter introduces the following topics:

- LINQ
- LINQ to XML
- LINQ to SQL

LINQ Overview

LINQ is a set of standard query operators that brings powerful query facilities right into the .NET Framework language such as C# and VB.NET. The LINQ framework brings together the capability of data access with the power of data manipulation. This section provides an overview of the capabilities of LINQ and the standard query operators, but Chapters 3 and 4, respectively, will discuss in great detail the LINQ query operators and language features that contribute to LINQ's direct, declarative style of queries.

The term Language Integrated Query signifies that the standard query facilities are architected directly into the developer's .NET-supported programming language of choice. These query facilities, known as the standard query operators, expose general-purpose query mechanisms that can be applied to

many facets of information, such as in-memory constructs as well as information retrieved from external sources such as relational data or XML.

These operators provide the capability to express query operations directly and declaratively within any .NET-based programming language. What makes all of this possible is the simple application of the query operators to an `IEnumerable<T>` source of information.

Found in the `System.Collections.Generic` namespace, the `IEnumerable<T>` interface, a new addition in version 2.0 of the .NET Framework, supports a simple iteration over a collection of a given (specified) type. The `IEnumerable<T>` interface provides a slick mechanism to iterate through an arbitrary collection of strongly typed objects using the C# `foreach` statement or the Visual Basic `FOR EACH` statement. To utilize the `foreach` semantics, this interface must be implemented.

So the question is, what does this mean for LINQ? It means that a query that implements this interface can be a source for the corresponding query expression. You saw several examples of this at the beginning of this chapter, and the best way to understand the LINQ technology is to see it in action.

The following example utilizes LINQ, a few standard query operators, and the `IEnumerable<T>` interface to query and process the contents within a defined array:

```
private void ShowLINQ()
{
  string[] firstnames = { "Scott", "Steve", "Ken", "Joe", "John",
                        "Alex", "Chuck", "Sarah"};

  IEnumerable<string> val = from fn in firstnames
                            where fn.StartsWith("S")
                            select fn;

  foreach (string name in val)
  {
    Console.WriteLine(name);
  }
}
```

The first statement defines an array of first names. This should not be new to any developer. The next statement, however, is new. A local variable, `val` in this case, is initialized with a Language Integrated Query expression. The query expression contains two query operators taken from plethora of standard query operators. In this example, two operators are used: `where` and `select`. The local variable `val` exposes the `IEnumerable<string>` interface, which provides the capability to iterate through the collection. The results are actually created as you start to iterate through them via the `foreach` statement.

From here, the query can be modified to add sorting or additional filtering as well as many other options, but that will be expanded on in later chapters. For now, suffice it to say that via LINQ you can query various source data types, such as XML and relational data, through a standard and consistent query model and related query operators.

To illustrate this, let's modify the directory example from earlier in this chapter. One of the great things about LINQ is that it enables you easily to ''map'' object-oriented objects within your .NET programming language to a database and the objects within a relational database. That means you can access those relational objects in a strongly typed, object-oriented manner.

To do this, a mapping to the database needs to be made, and that is accomplished by creating and declaring two classes. Those classes map the relational objects into the object-oriented world. The first class maps the actual database:

```
 [Database(Name="AdventureWorks")]
public class AdventureWorks : DataContext
{
    public AdventureWorks(string connection) : base(connection) {}
    public Table<DirectoryInformation> DirectoryInformation;
}
```

The second class maps the table and columns of the table you want to access:

```
 [Table(Name="DirectoryInformation")]
public class DirectoryInformation
{
    [Column(DbType="varchar(50)")]
    public string DirectoryName;

    [Column(DbType = "varchar(255)")]
    public string DirectoryDescription;
}
```

The class name maps to the table in the database you want to access, and the columns are mapped by adding metadata to a couple of variables.

This example is just to whet your appetite. There's not a lot of explanation here because there are more than a handful of chapters that discuss object mapping and querying in much greater detail.

Once the mapping is complete, the data can be queried. And not just queried, but queried using strongly typed syntax.

The first line of the following code accesses the database as an object, creating a new instance of the class previously defined, a strongly typed connection. Once you have the connection, you can access the table and data in a strongly typed fashion, as shown in the second and third lines. Notice that the columns in the table are accessed via dot notation directly in C#.

```
AdventureWorks db = new AdventureWorks("Integrated Security=sspi");

foreach (var item in db.DirectoryInformation)
    listBox1.Items.Add(item.DirectoryName + "  " +
item.DirectoryDescription);
```

Executing this code returns the data from the `DirectoryInformation` table and lists both the directory name and description in a list box.

To make it more interesting, take the directory example from the beginning of the chapter and modify it to join to this query. You'll recall that the code in the earlier example simply queried the `DirectoryInfo` class to return the directories on your local C drive. Combining it with this query, you join the `Name` property of the `DirectoryInfo` class to the `DirectoryName` column from the `DirectoryInformation`

table to return the `DirectoryDescription` information from the table. Just add the following highlighted code to the earlier query:

```
DirectoryInfo di = new DirectoryInfo("C:\\");

var query =
    from dir in di.GetDirectories()
    orderby di.Name
    select new
    {
        dir.Name,
        DirectoryDescription = (
        from d in db.DirectoryInformation
        where d.DirectoryName == di.Name
        select d.DirectoryDescription).FirstOrDefault()
   };

foreach (var item in query)
    listBox1.Items.Add(item.Name + "  " + item.DirectoryDescription);
}
```

To run this example, you first need to create a table in a database. The example used the AdventureWorks database and the following code to create the table:

```
CREATE TABLE [dbo].[DirectoryInformation](
    [DirectoryName] [varchar](50) NULL,
    [DirectoryDescription] [varchar](255) NULL
) ON PRIMARY

GO
```

You can use the following `INSERT` statement to add data to the `DirectoryInformation` table:

```
INSERT INTO DirectoryInformation (DirectoryName, DirectoryDescription)
    VALUES ('Windows', 'My Windows Directory')
GO
```

Before continuing, think about the amount of code you would have had to write to accomplish the same type of query in pre-LINQ technology. In the space of about two dozen lines, you were able to access data, query that data, and loop through that data simply and efficiently. In other technologies, you would have had to create a connection to the database, create an instance of a `SqlCommand` object and any other objects needed to execute a query, and write T-SQL code in your .NET code enclosed in quotation marks And that's not to mention all the work that has to be done once you get the data back—casting to the appropriate data types, and so on.

The good news is that LINQ does all of that for you. Sweet! And we haven't even covered XML yet.

Standard Query Operators

The LINQ standard query operators make up an API that provides the means of querying various data sources, such as arrays, collections, and even XML and relational data. They are a set of methods that are implemented by each specific LINQ provider (LINQ to SQL, LINQ to XML, LINQ to Objects,

and so on). The operators form a LINQ query pattern that operates on a sequence (an object that implements the `IEnumerable<T>` or `IQueryable<T>` interface).

There are two sets of standard query operators—one operates on objects of the `IEnumerable<T>` type and the other operates on objects of the `IQueryable<T>` type. The operators are made up of methods that are static members of the `Enumerable` and `Queryable` classes, allowing them to be called using static method syntax of instance method syntax. You will learn all about this in Chapter 4.

The standard query operators can be categorized by their operation ''type.'' For example, there are aggregate, projection, ordering, and grouping operators, among others. Take a look again at one of the examples used earlier in the chapter (repeated here for your convenience):

```
private void ShowLINQ()
{
  string [] firstnames = { "Scott", "Steve", "Ken", "Joe", "John",
                      "Alex", "Chuck", "Sarah"};

  IEnumerable<string> val = from fn in firstnames
                            where fn.StartsWith("S")
                            select fn;

  foreach (string name in val)
  {
    Console.WriteLine(name);
  }
}
```

The actual LINQ query is the middle part:

```
val = from fn in firstnames
  where fn.StartsWith("S")
  select fn;
```

In this example, several query operators are utilized from different operation types. The `select` query operator falls into the category of projection operators, and performs a projection over a sequence, an object that implements the `IEnumerable<T>` for a given type. In this case, the `select` operator enumerates the source sequence of first names.

```
select fn;
```

The `where` query operator is of the restriction operator type—in fact, it is the only operator of that type. Just like T-SQL, the LINQ `where` operator filters a sequence. In the preceding example, it filters the sequence by limiting the results returned to only those whose name begins with the letter *S*.

```
where fn.StartsWith("S")
```

If you are trying this example out yourself, create a new Windows Forms project and place a list box on the form. In the `Load` event of the form, place the following code:

```
string [] firstnames = { "Scott", "Steve", "Ken", "Joe", "John",
                      "Alex", "Chuck", "Sarah"};
```

```
IEnumerable<string> val = from fn in firstnames
                          where fn.StartsWith("S")
                          select fn;

foreach (string name in val)
{
  listbox1.Items.Add(name);
}
```

Press F5 to run the app. When the form loads and is displayed, the list box should be populated with the names of Scott, Steve, and Sarah. Now try changing the `where` clause, change the capital *S* to a lowercase *s* and rerun the app. Do you get results? Why not? If you haven't figured out why, Chapter 3, ''LINQ queries,'' will explain it.

Chapters 3 and 4 go deeper into LINQ and the standard query operators, so don't worry about understanding everything there is to know about LINQ just yet. This section was simply to whet your appetite. The following sections discuss LINQ to XML, which uses LINQ to query XML data, and LINQ to SQL, which uses LINQ to query relational data.

LINQ to XML Overview

LINQ to XML, or XLINQ, is the XML integration of the Language Integrated Query. LINQ to XML utilizes the standard query operators to provide the ability to query XML data. Also at your disposal are operators that provide functionality akin to XPath, letting you navigate up and down and navigate XML tree nodes such as descendants and siblings seamlessly and efficiently.

If you have ever used, and disliked, the DOM, you will love LINQ to XML. The great thing about LINQ to XML is that it provides a small-footprint, in-memory version of the XML document that you are querying. LINQ to XML utilizes the XML features of the `System.Xml` namespace, specifically the reader and writer functionality exposed by the `System.Xml` namespace.

LINQ to XML exposes two classes that help LINQ integrate with XML: `XElement` and `XAttribute`. The `XElement` class represents an XML element and is used in LINQ to XML to create XML element nodes or even to filter out the data you really care about. `XElement` ties itself to the standard query operators by enabling you to write queries against non-XML sources and even persist that data to other sources.

The `XAttribute` class is a name/value pair associated with an XML element. Each `XElement` contains a list of attributes for that element, and the `XAttribute` class represents an XML attribute. Within LINQ, both the `XElement` and `XAttribute` types support standard syntax construction, meaning that developers can construct XML and XML expressions using the syntax that they already know.

The following example uses the `XElement` to construct a simple XML document. The first `XElement` defines the outer node while the two inner `XElement` parameters define the two inner nodes of `FirstName` and `LastName`.

```
var x = new XElement("Employee",
            new XElement("FirstName", "Scott"),
            new XElement("LastName","Klein"));

var s - x.ToString();
```

Here are the results of this code:

```
<Employee>
  <FirstName>Scott</FirstName>
  <LastName>Klein</LastName>
</Employee>
```

You'll notice the use of `var` in the previous example. The `var` keyword tells the compiler to infer the type of the variable from the expression on the right side of the statement. The `var` keyword will be discussed in detail in Chapter 2, ''A Look at Visual Studio 2008''.

Also notice in the previous example how much easier the code is to read. The code actually follows the structure of an XML document, so you can see what the resulting XML document will look like.

The next example uses the `XAttribute` type to add an attribute to the XML:

```
var x = new XElement("Employee",
            new XAttribute("EmployeeID", "15"),
            new XElement("FirstName", "Scott"),
            new XElement("LastName","Klein"));

var s - x.ToString();
```

And here are the results from it:

```
<Employee Employee="15">
  <FirstName>Scott</FirstName>
  <LastName>Klein</LastName>
</Employee>
```

While the capability to easily define the contents of the XML is cool, the real power comes from the ability to pass an argument that is not user-defined but in reality comes from an outside source, such as a query, which can be enumerated and turned into XML via the standard query operators. For example, the following takes the array of names from the first example and uses that as the source of the query for which to construct XML:

```
string [] firstnames = { "Scott", "Steve", "Ken", "Joe", "John",
                         "Alex", "Chuck", "Sarah"};

var r = new XElement("Friends",
            from fn in firstnames
            where fn.StartsWith("S")
            select new XElement("Name", fn))
textbox1.Text = rToString();
```

Here are the results from this code:

```
<Friends>
  <Name>Scott</Name>
  <Name>Steve</Name>
  <Name>Sarah</Name>
</Friends>
```

This isn't to say that I only have friends whose first names begin with the letter *S*, but you get the idea. This query returns a sequence of `XElements` containing the names of those whose first name begins with the letter *S*. The data comes not from a self-generated XML document but an outside source, in this case the array of first names. However, the data could just as easily come from a relational database or even another XML document.

What `XElement` enables you to do is query non-XML sources and produce XML results via the utilization of the `XElements` in the body of the `select` clause, as shown earlier. Gnarly.

The object of these simple examples is to illustrate the basic concepts of LINQ to XML and the great power, flexibility, and ease with which XML can be manipulated. Note that the same standard query operators were used to generate the XML document in this example as in the first one. Nothing had to be changed in the query really, other than using the types to help integrate LINQ with XML to build the resulting XML. Yet the query operators remained the same, as did the overall syntax of the query expression. This way is much better than trying to figure out XQuery or XPath, working with the DOM or even XSLT. Chapters 10 through 13 cover LINQ to XML in much greater detail.

LINQ to SQL Overview

LINQ to SQL, or DLINQ, is another component in the LINQ technology ''utility belt.'' It provides a mechanism for managing relational data via a run-time infrastructure. The great thing about this is that LINQ still keeps its strong points, such as the ability to query. This is accomplished by translating the Language Integrated Query into SQL syntax for execution on the database server. Once the query has been executed, the tabular results are handed back to the client in the form of objects that you as a developer have defined.

If you have been following the LINQ talk, you already know that LINQ to SQL is the next version of ADO.NET. This is great news, and by the time you are done with this section and the section on LINQ to SQL later in the book, you will surely know why. LINQ takes advantage of the information produced by the SQL schema and integrates this information directly into the CLR (Common Language Runtime) metadata. Because of this integration, the definitions of the SQL tables and views are compiled into CLR types, making them directly accessible from within your programming language.

For example, the following defines a simple schema based on the `Person.Contact` table from the AdventureWorks database:

```
[Table(Name="Person.Contact")]
public class Contact
{
  [Column(DBType = "nvarchar(50) not null")]
  public string FirstName;
  [Column(DBType = "nvarchar(50) not null")]
  public string LastName;

  [Column(DBType = "nvarchar(50) not null")]
  public string EmailAddress;

}
```

Once this schema is defined, a query can be issued. This is where LINQ comes in. Using the standard query operators, LINQ translates the query from its original query expression form into a SQL query for execution on the server:

```
private void button5_Click(object sender, EventArgs e)
{
  DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
    Security=sspi");

  Table<Contact> contact = context.GetTable<Contact>();

  var query =
      from c in contact
      select new { c.FirstName, c.LastName, c.EmailAddress} ;

  foreach (var item in query)
  listBox1.Items.Add(item.FirstName + " " + item.LastName + " " +
    item.EmailAddress);

}
```

Following are partial results from the query:

```
gustavo Achong gustavo0@adventure-works.com
catherine0@adventure-works.com
kim2@adventure-works.com
humberto0@adventure-works.com
pilar1@adventure-works.com
frances0@adventure-works.com
margaret0@adventure-works.com
carla0@adventure-works.com
jay1@adventure-works.com
```

Obviously there is much more to LINQ to SQL, but the examples here illustrate what it can do and the basic features and fundamental concepts of LINQ to SQL.

If you were to query this table via SQL Query Analyzer or SQL Server Management Studio, you'd know that the `Person.Contact` table in the AdventureWorks database is 28 rows shy of 20,000, so the preceding list is only the first nine, but you get the idea. How would you filter this query to return only a specific few rows?

Typically I like to wait until the third or fourth chapter to start handing out ''homework assignments,'' but with the background presented in this chapter you should be able to figure this out quite easily. The `Person.Contact` table has some additional columns that you can use to filter the results. For example, it has a column named `Title`, which contains values such as ''Mr.'' and ''Ms.'' It also has a column named `EmailPromotion`, an `int` datatype with values of `0` through `2`.

Your exercise for this chapter is to filter the query on either the `Title` column or the `EmailPromotion` column, using a standard query operator, so that the results returned are much less that 20,000. FYI if you are going to use the `Title` column: some of values of the column are null, so don't query where `Title` is null.

The goal of LINQ to SQL and its related tools is to drastically reduce the work of the database developer. Chapters 9–13 will discuss LINQ to SQL in much more depth.

Summary

This chapter introduced you to the LINQ project, a set of .NET Framework extensions that extend the C# and Visual Basic .NET programming languages with a native query language syntax that provides standard query, set and manipulation operations.

This chapter began by discussing LINQ and the set of standard query operators that is a combination of SQL query capabilities with the power and flexibility of data manipulation. From there, the topics of LINQ to XML and LINQ to SQL were discussed, which take the power and flexibility of LINQ and apply it to the querying of relational data and XML documents using the same syntax provided by LINQ.

With this foundation, the next chapter will take a look at the next release of Visual Studio by looking at the specific features LINQ supports for Visual Basic 9.0 and C#.